Reitinhakualgoritmien käyttö ja soveltaminen videopelituotannossa

Turun yliopisto
Tietotekniikan laitos
TkK-tutkielma
Tietotekniikka
Joulukuu 2025
Aaron Finnilä

Turun yliopiston laatujärjestelmän mukaisesti tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck -järjestelmällä.

TURUN YLIOPISTO
Tietotekniikan laitos

Aaron Finnilä: Reitinhakualgoritmien käyttö ja soveltaminen videopelituotannossa

TkK-tutkielma, 22 s.
Tietotekniikka
Joulukuu 2025

Reitinhaku on usean vuosikymmenen ollut laajasti käsitelty ongelma muun muassa karttasovelluksissa, robotiikassa ja videopeleissä. Ongelmassa keskitytään siihen, miten paikasta A päästään paikkaan B mahdollisimman nopeasti. Ratkaisuna tähän ongelmaan on kehitetty erilaisia reitinhakualgoritmeja, jotka voivat erittäin nopeasti löytää lyhimmän tai ainakin melkein lyhimmän reitin. Eri algoritmeilla on kuitenkin hyvin erilaiset lähestymistavat ongelmaan. Tämän takia oikean algoritmin valitseminen kyseessä olevaan sovelluskohteeseen on kriittistä reitinhaun tehokkuuden kannalta. Tehokas reitinhaku on keskeinen tekijä hyvälle käyttäjäkokemukselle karttasovelluksissa ja videopeleissä. Robotiikassa ja autonomisissa järjestelmissä tehokkaat reitinhakualgoritmit parantavat järjestelmien toiminnallisuutta ja koettua älykkyyttä. Tutkielmassa käsitellään viimeisen viiden vuoden tutkimuskirjallisuuden kannalta olennaisia algoritmeja, ja miten näitä sovelletaan videopeleihin. Voidaan havaita, että uudemmat algoritmit voivat oikein toteutettuina olla huomattavasti tehokkaampia kuin vanhemmat vaihtoehdot. Voidaan myös havaita, että oikean mallintamistekniikan ja heuristiikan valitseminen on ratkaisevaa sovelluskohteen suorituskyvyn ja toiminnallisuuden kannalta.

Asiasanat: algoritmi, reitinhaku, videopelituotanto, heuristiikka, mallintaminen

Sisällys

1 Johdanto **1**

2 Reitinhakualgoritmit videopeleissä **4**

2.1 Informoidut ja epäinformoidut algoritmit 4

2.2 A*-algoritmi 6

2.3 A*-algoritmin variaatiot 7

2.4 Vahvistusoppimiseen perustuvat algoritmit 10

3 Reitinhakualgoritmien soveltaminen videopeleihin **12**

3.1 Yleisimmät mallintamistekniikat 12

3.2 Heuristiikka 16

3.3 Syväoppimisen käyttö heuristiikan määrittämisessä 18

4 Yhteenveto **21**

Lähdeluettelo **23**

1 Johdanto

Reitinhaku tarkoittaa tietokoneen suorittamaa hakua, jonka tarkoituksena on etsiä mahdollisimman hyvä reitti kahden pisteen välillä. Reitinhakualgoritmeja käytetään laajasti videopeleissä, robotiikassa ja karttasovelluksissa. Ne auttavat sovelluksia ja autonomisia järjestelmiä toimimaan sulavasti ja tehokkaasti, ja niiden hyvä toteutus on usein kriittistä miellyttävän käyttäjäkokemuksen kannalta. Videopeleissä reitinhakualgoritmit saavat erilaiset hahmot vaikuttamaan älykkäiltä ja mahdollistavat navigoinnin erittäin monimutkaisissa ympäristöissä. Ne luovat peliin immersiota eli kokemukseen uppoutumista. Robotiikassa algoritmit kertovat robotille mihin mennä, ja auttavat sitä minimoimaan törmäyksiä [1]. Karttasovelluksissa kehittyneet reitinhakualgoritmit voivat kertoa käyttäjälle, kauanko jokin matka kestää, ja mikä on nopein reitti. Usein käyttäjä voi myös valita, mitä haluaa priorisoida. Reitin hyvyys riippuu sille annetuista kriteereistä. Esimerkiksi karttasovelluksissa tämä tarkoittaa usein nopeinta reittiä, mutta voi myös tarkoittaa polttoaineellisesti halvinta reittiä tai lyhintä reittiä. Reaaliaikaisissa GPS-sovelluksissa painotetaan usein laskennan nopeutta reitin optimaalisuuden sijaan, sillä käyttäjän on tärkeä saada nopeasti tietää mihin suuntaan kannattaa mennä [2].

Tämä kandidaatintutkielma on toteutettu kirjallisuuskatsauksena. Tutkielmassa keskitytään tarkemmin reitinhakualgoritmeihin videopelien näkökulmasta. Tavoitteena on selvittää, millaisia algoritmeja alan modernissa tutkimuskirjallisuudessa ilmenee. Lisäksi yritetään selvittää, miten näitä algoritmeja sovelletaan videope-

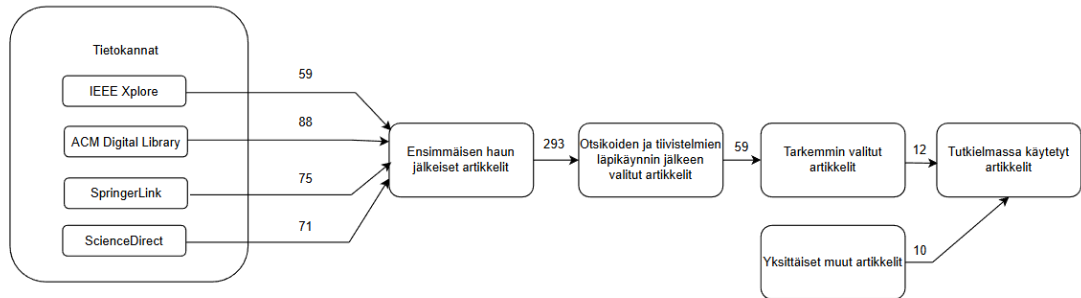
leihin. Näiden tavoitteiden pohjalta muotoutuvat kaksi tutkimuskysymystä, joihin pyritään vastaamaan tutkielman aikana:

TK1 Millaisia reitinhakualgoritmeja esiintyy viimeisen viiden vuoden aikana julkaistussa videopelituotantoa käsittelevässä tutkimuskirjallisuudessa?

TK2 Miten reitinhakualgoritmeja sovelletaan videopeleihin viimeisen viiden vuoden aikana julkaistussa tutkimuskirjallisuudessa?

Tutkielmaa varten on käyty läpi kirjallisuutta neljästä eri tietokannasta: IEEE Xplore, ACM Digital Library, SpringerLink ja ScienceDirect. Haku on toteutettu englanniksi, sillä alan kirjallisuus on laajalti englanniksi. Hakua on rajattu sisältämään tuloksia vuosilta 2020-2025, ja haussa on käytetty seuraavanlaista hakulauseketta: ”pathfinding*” AND (“videogame*” OR “video game*”). Lausekkeessa on käytetty jokerimerkkiä *, joka tarkoittaa mitä tahansa merkkiä. Esimerkiksi videogame* tarkoittaa, että sana voi olla mm. videogame tai videogames. Lausekkeessa on myös käytetty boolean-operaattoreita, joiden avulla yritetään varmistaa, että tulokset liittyvät sekä reitinhakuun että videopeleihin.

Kuva 1.1 havainnollistaa tiedonhaun prosessia. Lähteitä on prosessissa vaiheittain karsittu pois. Aluksi lähteitä oli 293. Näistä valittiin jatkoon hyödyllisiltä ja aiheeseen kuuluvilta vaikuttavat. Tässä vaiheessa valinta tehtiin otsikon ja tiivistelmän perusteella. Tämän jälkeen lähteitä oli 59. Näitä karsittiin sitten tarkemmin artikkelin saatavuuden, kielen ja sisällön perusteella. Prosessin jälkeen artikkeleita oli 12. Kyseisen prosessin läpikäyneiden artikkelien lisäksi on lähteinä käytetty myös yksittäisiä muita artikkeleita. Näitä artikkeleita on käytetty lähinnä taustatietoa varten. Muita artikkeleita on tässä tutkielmassa käytetty 10.



Kuva 1.1: Tiedonhaun prosessi.

Tutkielman toisessa luvussa käsitellään reitinhakua videopeleissä yleisellä tasolla ja sitä, millaisia reitinhakualgoritmeja viime vuosien aikana julkaistussa aiheen tutkimuskirjallisuudessa ilmenee. Kolmannessa luvussa käydään läpi, miten näitä algoritmeja sovelletaan eri peliympäristöihin. Viimeiseksi on yhteenveto, jossa vastataan tutkimuskysymyksiin ja pohditaan millaisia päätelmiä tutkielman tulosten perusteella voi tehdä. Mietitään myös lyhyesti mitä tulevien vuosien tutkimus reitinhakualgoritmien käyttämisestä videopelituotannossa voisi pitää sisällään.

2 Reitinhakualgoritmit videopeleissä

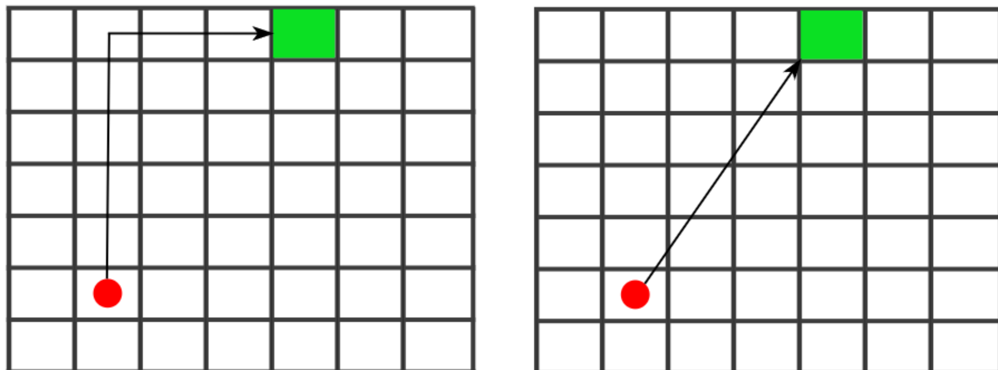
Reitinhaku on pitkään ollut keskeinen ongelma videopelituotannossa. Ongelmassa yritetään selvittää, miten videopelissä olevan hahmon saa valitsemaan parhaan mahdollisen reitin navigoidessaan paikasta A paikkaan B. Reitinhakuongelmaa käsiteltiin pitkään ennen kuin se ilmeni videopelituotannossa. Vuonna 1959 julkaistu Dijkstran algoritmi ilmeni graafiteorian yhteydessä. Sitä pidetään usein ensimmäisenä reitinhakualgoritmina. Yhdeksän vuotta myöhemmin julkaistiin A*-algoritmi, joka on parannettu versio Dijkstran algoritmista. A*-algoritmia käytetään vielä laajalti modernissa videopelituotannossa, ja se on eniten käytetty reitinhakualgoritmi tekoälysovelluksissa [3]. Lisäksi monet uudemmista reitinhakualgoritmeista ovat muokattuja versioita siitä.

Jotta reitinhakualgoritmeja voidaan käyttää videopeleissä, täytyy pelin avaruutta mallintaa jollain tavalla. Tätä prosessia käsitellään laajemmin luvussa 3. Tässä luvussa keskitytään itse algoritmeihin ja siihen, miten ne eroavat toisistaan. Seuraavassa osiossa käsitellään informoituja ja epäinformoituja algoritmeja. Luvun toisessa ja kolmannessa osiossa keskitytään A*-algoritmiin ja sen variaatioihin, ja viimeisessä osiossa käsitellään vahvistusoppimiseen perustuvia algoritmeja.

2.1 Informoidut ja epäinformoidut algoritmit

Laajasti käsiteltynä on olemassa kahdenlaisia reitinhakualgoritmeja: informoituja ja epäinformoituja. Informoiduissa algoritmeissa haulla on jonkinlainen käsitys siitä,

missä suunnassa määränpää sijaitsee lähtöpisteestä. Tämän avulla voidaan välttää ylimääräistä työtä ja olla menemättä liian pitkälle väärään suuntaan. Epäinformoiduissa algoritmeissa haku ei tiedä ollenkaan, missä suunnassa määränpään kuuluisi olla. Dijkstran algoritmi on esimerkki epäinformoidusta hausta. Oletetaan esimerkiksi, että haku toteutetaan kahden ulottuvuuden tilassa, joka koostuu solmuista. Tilaa voidaan myös kuvailla ruudukoksi. Dijkstran algoritmi etsii tällaisessa tilassa määränpäätä sokeasti ja tasaisesti joka suunnasta, kunnes lopulta löytää sen. Informoidut algoritmit, kuten A*-algoritmi, toimivat eri lailla. Ne käyttävät heuristiikkaa havaitakseen, missä suunnassa määränpään kuuluisi olla. A*-algoritmissa käytetään yleisesti kahta erilaista heuristiikkaa: Manhattan-etäisyyttä ja Euklidista etäisyyttä. Manhattan-etäisyys tarkoittaa korttelityyppistä etäisyyttä. Euklidinen etäisyys tarkoittaa ns. linnuntie-etäisyyttä eli suoraa etäisyyttä pisteestä A pisteeseen B. Kuva 2.1 havainnollistaa näitä heuristiikkoja. [2]



Kuva 2.1: Manhattan-etäisyys (vasen) ja Euklidinen etäisyys (oikea).

Alan modernissa tutkimuskirjallisuudessa ilmenee vielä usein sekä A*- että Dijkstran algoritmia. A*-algoritmi on kuitenkin yleisempi, ja kuten aiemmin on mainittu monet uudemmista reitinhakualgoritmeista ovat muokattuja versioita siitä.

2.2 A*-algoritmi

Jotta voitaisiin paremmin ymmärtää A*-algoritmin variaatioita ja niiden eroavaisuuksia, täytyy ensiksi tarkastella perinteisen A*:n toimintaa syvemmin. Oletetaan yksinkertaisuuden vuoksi algoritmin ympäristön olevan kahden ulottuvuuden ruudukko, jossa voidaan liikkua pelkästään vaaka- ja pystysuunnassa.

A*-algoritmin toteutuksessa puhutaan avatuista ja suljetuista solmuista. Avatut solmut ovat solmuja, joita algoritmi tutkii aktiivisesti. Suljetut solmut ovat solmuja, jotka algoritmi on jo tutkinut. Algoritmi alkaa sillä, että avataan aloitussolmu a ja lasketaan sen arvo *f(a)*. Solmu laitetaan sitten eräänlaiseen jonoon, *prioriteettijonoon*, jossa jonossa olevat solmut lajitellaan niiden *f(n)* arvon mukaan, jossa n on jokin jonossa oleva solmu. *f(n)*-arvo on solmun tällä hetkellä laskettu lyhin etäisyys aloitussolmusta lisättynä sen arvioituun etäisyyteen maalisolmusta. Prioriteettijonosta valitaan sitten solmu, jonka *f(n)*-arvo on pienin. Tämän jälkeen tarkistetaan, onko solmu n maalisolmu, eli onko etäisyys maaliin nolla. Jos ei, niin suljetaan kyseinen solmu ja avataan sen naapurisolmut, eli vieressä olevat solmut. Sama prosessi jatkuu sitten avatuilla solmuilla, kunnes maalisolmu löytyy. [4]

Algoritmissa tutkitaan siis ensiksi aloitussolmua, sitten useampia ja useampia naapurisolmuja maalisolmun arvioidun etäisyyden eli heuristiikan avustamana. Avattuja solmuja vertaillaan siis jatkuvasti keskenään, ja heuristiikan avulla priorisoidaan niitä, jotka vaikuttavat lupaavimmilta. Algoritmissa pidetään myös kirjaa siitä, että minkä solmun kautta mihinkin solmuun on päästy. Tämän avulla voidaan jäljittää reitti, jonka kautta on lopulta päästy maalisolmuun. Alkuperäisessä A*-algoritmin julkaisussa [4] mainitaan, että löytääkseen aina parhaan mahdollisen reitin täytyy heuristiikan olla ei-yliarvioiva. Toisin sanoen heuristiikka ei saa antaa uskoa, että maalisolmu olisi kauempana kuin mitä se on. Heuristiikkaa ja sen merkitystä algoritmeissa käsitellään tarkemmin luvussa 3.

2.3 A*-algoritmin variaatiot

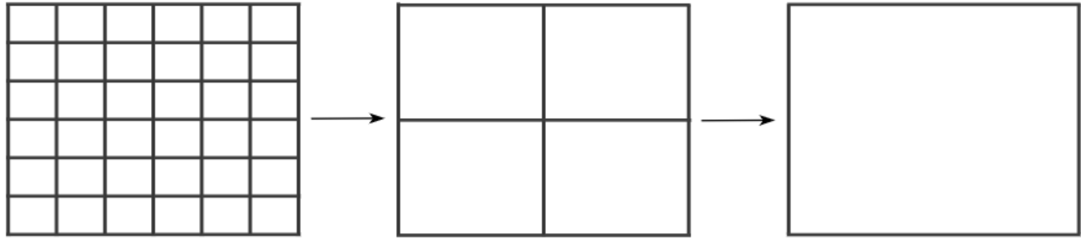
LRA* (Local Repair A*) on muokattu versio A*-algoritmista. Sitä käytetään ns. moniagenttiympäristöissä (engl. multi-agent environment). Tällaisessa ympäristössä on monta eri hahmoa jotka toteuttavat reitinhakua. Perinteinen A*-algoritmi toimii luotettavasti pelkästään staattisessa ympäristössä, jossa sen toteuttaja on ainoa liikkuva hahmo. Tämän takia moniagenttiympäristöä varten täytyy valita jokin toinen tapa toteuttaa reitinhakua. LRA* on yksi vaihtoehto. Sitä on käytetty laajasti videopelituotannossa [5], ja sitä ilmenee yhä alan kirjallisuudessa [3]. Algoritmi toimii käytännössä samalla tavalla kuin perinteinen A*, mutta se reagoi konfliktin tapahtuessa. Eli jos kaksi eri videopelihahmoa toteuttaa reitinhakua LRA*-algoritmilla, ja jossain vaiheessa toteutusta hahmot törmäävät, niin reitinhaku toteutetaan uudelleen ja jatketaan siitä, missä törmäys tapahtui. [3]

Törmäyksiä ei siis LRA*-algoritmin tapauksessa yritetä estää. Tämä voi säästää muistia ja laskentatehoa, sillä muiden agenttien muuttuvia sijainteja ei tarvitse jatkuvasti tarkkailla. Tällä algoritmilla ilmenee kuitenkin usein ongelmia monimutkaisemmissa ympäristöissä [3]. Toinen vaihtoehto moniagenttiympäristössä toteutettavaan reitinhakuun on HPA* (Hierarchical Path-Finding A*). Yksi heikkous joka ilmenee perinteisessä A*-algoritmissa on vaaditun laskentatehon kasvu pelitilan suurentuessa. HPA*-algoritmi helpottaa tätä ongelmaa jakamalla tilan pienempiin osiin, ja tekemällä reitinhaun näille pienemmille alueille. Lisäksi algoritmi vähentää vaadittua reaaliaikaista laskentatehoa tekemällä optimaaliset alueiden väliset reitit etukäteen. HPA*-algoritmi ei siis ole optimaalinen, eli se ei voi taata löytävänsä lyhimmän mahdollisen reitin. Se voi kuitenkin olla jopa 10 kertaa nopeampi itse reitin löytämisessä verrattuna tavalliseen A*-algoritmiin ja löytää reittejä, jotka ovat korkeintaan yhden prosentin optimaalista reittiä pidempiä. [6]

Algoritmi uhraa siis optimaalisuutensa parantaakseen laskennallista tehokkuutta ja suoritusaikaa. HPA*-algoritmi toimii myös dynaamisissa ympäristöissä. Dy-

naamisella ympäristöllä voidaan tarkoittaa mm. moniagenttiympäristöä tai ympäristöä, jossa pelitila voi muuttua. Esimerkiksi jos pelissä räjähtää silta, jonka yli hahmo oli menossa, täytyy reitinhaku toteuttaa uudelleen. Tällaisessa tilanteessa HPA*-algoritmi on tehokas, sillä se voi tehdä reitinhaun uudelleen nykyiselle, lokaalille alueelle. Sen ei siis tarvitse tehdä koko pelitilan hakua uudelleen vaan pelkästään rajoitetun tilan, jossa muutos tapahtui. Jos taas tulee törmäys toisen hahmon kanssa, niin HPA* tekee nopeasti uuden reitin. Tämä uusi reitti tehdään rajoitetulla tarkkuudella, jotta saataisiin hahmo nopeasti liikkelle keskittymättä liikaa reitin optimaalisuuteen. Hahmo siis ohjataan suurin piirtein oikeaan suuntaan, ja reittiä parannetaan myöhemmin jos tarpeellista. [6]

HPA*-tyyppistä algoritmia voi myös mahdollisesti käyttää 3D-reitinhaussa [7]. Reittien laskemisella etukäteen on kuitenkin haittapuolia. Eri alueiden ja reittien tallentaminen vie paljon muistia. PRA*-algoritmilla (Partial Refinement A*) on erilainen lähestymistapa ongelmaan. Algoritmi alkaa sillä, että lähdetään korkealla tasolla oikeaan suuntaan. Haun edetessä optimoidaan reitin seuraavia tulevia osioita, mutta ei koko reittiä. Näin saadaan jaettua reitinhaun laskentatehon vaativuus tasaisemmin koko prosessille. Tämä tapahtuu hakualueen abstraktoinnin avulla [8], jota demonstroi kuva 2.2. Esimerkiksi jos matkustetaan eri maiden välillä, ei ole tarpeellista tietää jokaista matkalla olevaa katua heti alussa. Tarkastellaan mieluummin tämänhetkisen maan katuja ja kaupunkeja, ja vasta myöhemmin huolehditaan yksityiskohtaisesti muiden maiden alueista. Muun muassa RTS (Real-time Strategy) peleissä PRA*-tyyppisestä algoritmista voisi olla paljon hyötyä. [9]



Kuva 2.2: Hakualueen abstraktointi.

Artikkelissa [10] kerrotaan, miten PRA* ja HPA*-algoritmien yhdistelmää käytettiin onnistuneesti pelissä Dragon Age: Origins. Peli julkaistiin vuonna 2009, ja siinä käytettiin muunneltua versiota navmesh-mallintamisesta abstraktioiden luomista varten. Navmesh-mallintamista käsitellään lisää luvussa 3.

JPS-algoritmi (Jump-Point Search) on optimaalinen reitinhakualgoritmi joka perustuu A*-algoritmiin. JPS-algoritmissa nopeutetaan reitinhaun prosessia ohittamalla turhat solmut, ja hyppäämällä A*-algoritmin määrittämään lupaavimpaan suuntaan. Tämä säästää laskentatehoa ja aikaa, sillä algoritmin ei tarvitse tutkia sellaisia solmuja jotka ovat mitättömiä lopputuloksen kannalta. [11]

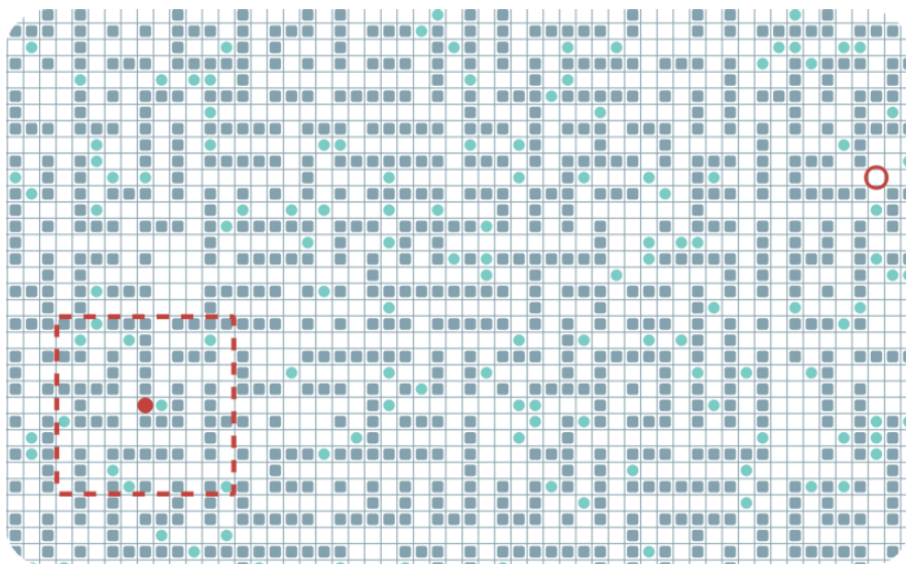
Perinteinen JPS-algoritmi toimii pelkästään ei-painotetuissa ruudukoissa. JPSW (Jump-Point Search Weighted) on JPS-algoritmistä kehitetty versio, joka toimii painotetuissa ruudukoissa [12]. JPS-algoritmista on myös kehitetty kolmen ulottuvuuden versio, JPS-3D, joka on viimeisintä tekniikkaa edustava 3D-reitinhakualgoritmi [7].

Artikkelissa [13] tutkitaan, miten syväoppivia menetelmiä voidaan käyttää reitinhaussa. Syväoppimista hyödynnettiin heuristiikan muodostamista varten. Prosessia käsitellään tarkemmin luvussa 3. Itse reitinhakua varten käytettiin Focal Search-algoritmia, joka on versio A*-algoritmista, jossa uhrataan optimaalisuus nopeampaa ja laskennallisesti halvempaa toteutusta varten. Focal Search-algoritmi eroaa muista

epäoptimaalisista algoritmeista siten, että siinä on rajattu, kuinka paljon optimaalisesta reitistä voidaan poiketa. Heuristiikkafunktiossa olevaa painoarvoa w muuttamalla voidaan säädellä sitä, miten paljon haun optimaalisuutta rajataan. Focal Search-algoritmi voi siis olla hyödyllinen, jos halutaan pysytellä mahdollisimman lähellä optimaalista tulosta, mutta kuitenkin tehostaa hakua.

2.4 Vahvistusoppimiseen perustuvat algoritmit

PO-MAPF (Partially observable multi-agent pathfinding) tarkoittaa reitinhakua, joka tapahtuu moniagenttiympäristössä, ja jossa agenteilla on rajattu tieto toisistaan. Suurin osa moniagenttiympäristöissä toimivista algoritmeista käyttää keskitettyä ohjaajaa (engl. central controller). Ohjaaja suunnittelee agenteille reittejä, joissa ei tapahdu törmäyksiä. Tämä tapahtuu ennen kuin agentit lähtevät liikkelle, ja tätä varten tarvitaan tieto kaikkien agenttien reiteistä ja määränpäistä. Artikkelissa [14] tieto agenttien välillä rajataan kuitenkin niin, että agentit tietävät ainoastaan toistensa sijainnit. Lisäksi agentit ovat ainoastaan tietoisia esteistä ja toisista agenteista jotka ovat heidän lähettyvillään. Kuva 2.3 havainnollistaa ongelmaa. [14]



Kuva 2.3: Agentin rajattu näkyvyys. Lähde: [14]. © 2024 IEEE

Agenttien täytyy siis tehdä yhteistyötä rajatulla tiedolla onnistuakseen tehtävissään, toisin sanoen päästäkseen omaan määränpäähänsä. Artikkelissa tästä käytetään esimerkkinä videopeliä Starcraft, jossa on rajattu näkyvyys pelin alueesta, ja jossa usean agentin täytyy tehdä yhteistyötä. Ongelma ratkaistiin artikkelissa kahdella tavalla. Ensimmäisessä käytettiin uudelleen suunnittelevaa heuristista hakualgoritmia RePlan, ja toisessa käytettiin syväoppivaa algoritmia EPOM. Näitä algoritmeja vertailtiin muiden samanlaisten algoritmien kanssa, ja tutkittiin myös miten käy jos vaihdellaan EPOM ja RePlan algoritmien välillä. Tulosten perusteella paras vaihtoehto kyseiseen ongelmaan oli vaihdella EPOM ja RePlan algoritmien välillä. [14]

3 Reitinhakualgoritmien soveltaminen videopeleihin

Kuten luvussa 2 mainittiin, pelin tilaa täytyy usein mallintaa, jotta reitinhakualgoritmeja voitaisiin hyödyntää. Yksinkertaisena esimerkkinä voi käyttää kahden ulottuvuuden peliä, joka on ylhäältäpäin kuvattu. Jotta tällaisessa pelissä voitaisiin käyttää reitinhakualgoritmia, täytyy tila jakaa solmuihin. Algoritmi voi sitten etsiä määränpäätä näistä solmuista, ja löytäessään määränpään alkaa liikuttamaan hahmoa sinne löydettyä reittiä pitkin. Tilan monimutkaistuessa ilmenee kuitenkin erilaisia haasteita. Muun muassa 3D-peleissä joissa hahmojen liike ei ole täysin pystysuunnassa rajoitettua on erityisen hankalaa mallintaa peliympäristöä reitinhakua varten [15]. Erilaisissa peliympäristöissä täytyy käyttää erilaisia tekniikoita mallintamiseen. Seuraavaksi käydään läpi yleisimmät tekniikat joita käytetään tätä prosessia varten. Luvun viimeisissä osioissa käsitellään heuristiikkaa ja syväoppimisen käyttöä heuristiikan määrittämisessä.

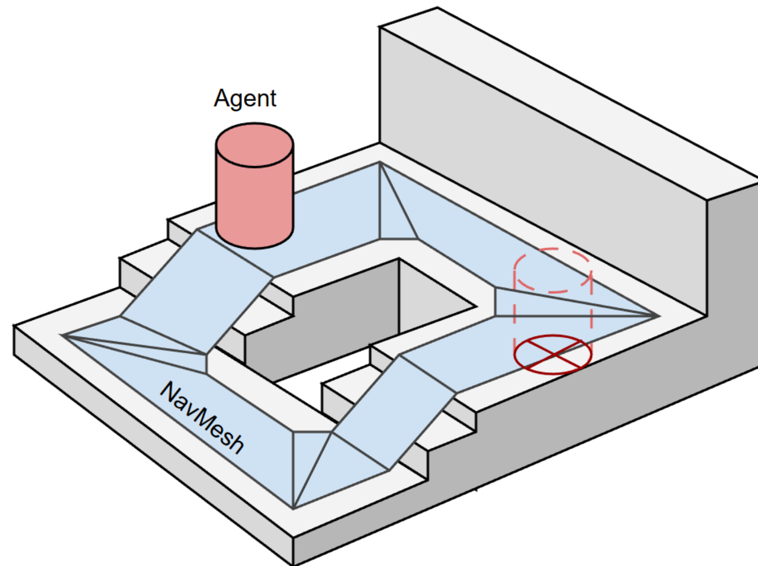
3.1 Yleisimmät mallintamistekniikat

Yksi yleisimmistä mallintamistekniikoista, jota on käytetty paljon historiallisesti ja jota käytetään yhä, on säännöllinen ruudukko (engl. regular grid). Säännöllisiä ruudukoita on käytetty laajasti videopelituotannossa [15], ja ne ovat yleinen ja tunnettu tapa mallintaa pelitilaa [16]. Alkeellisissa kahden ulottuvuuden peleissä neliön-

muotoiset solmut ovat tehokas ja kätevä tapa jakaa tilaa. Säännöllisiä ruudukoita on kuitenkin erilaisia. Ne voidaan karkeasti jakaa kahteen ryhmään: painotetut ja ei-painotetut ruudukot. Painotetuissa ruudukoissa eri solmujen välillä voi olla eri arvoja. Näitä solmujen välillä olevia yhteyksiä kutsutaan usein kaariksi (engl. edge). Esimerkkinä voi käyttää peliä, jossa on alueita, joiden läpi kulkee hitaammin kuin toisten. Hitaampia alueita mallintaville solmuille voisi asettaa suuremman kaaren pituuden. Tämän avulla reitinhakualgoritmi voisi huomioida alueiden väliset nopeuserot valitessaan sopivaa reittiä, ja siten parantaa reitinhakua. Painotettuja ruudukoita voi myös käyttää mallintaakseen ylämäkiä. Tämän voi tehdä esimerkiksi laittamalla kaaren pidemmäksi ylämäkeen, ja lyhyemmäksi alamäkeen. Hahmojen nopeuden hidastuminen ylämäkeen voi lisätä pelaajan koettua realismia.

Vuorostaan ei-painotetuissa ruudukoissa jokainen kaari on yhtä pitkä. Suosittu 80-luvulla julkaistu PacMan-peli on esimerkki pelistä, jonka pelitilan voisi mallintaa ei-painotettuna ruudukkona. Ruudukko koostuu pelkästään solmuista joissa voi kulkea, ja solmuista joissa ei voi kulkea. Pelitilan mallintaminen ei vaadi tässä tapauksessa monimutkaisempaa toteutusta.

Navigation mesh (navmesh) on modernimpi vaihtoehto säännölliselle ruudukolle videopelien mallintamisessa. 2000-luvulla suosioon noussut mallintamistekniikka perustuu tilan mallintamiseen monikulmioiden avulla. Toisin kuin säännöllisissä ruudukoissa, navmesh mallintaa pelkästään tilan jossa hahmot voivat liikkua. Tämä säästää laskennallista tehoa, sillä hahmojen ei tarvitse jatkuvasti tarkistaa tapahtuuko törmäyksiä. Navmesh:llä toteutettu mallinnus on myös usein tarkempi kuin säännöllisellä ruudukolla, sillä tila jakaantuu yleensä pienempiin osiin. [6]



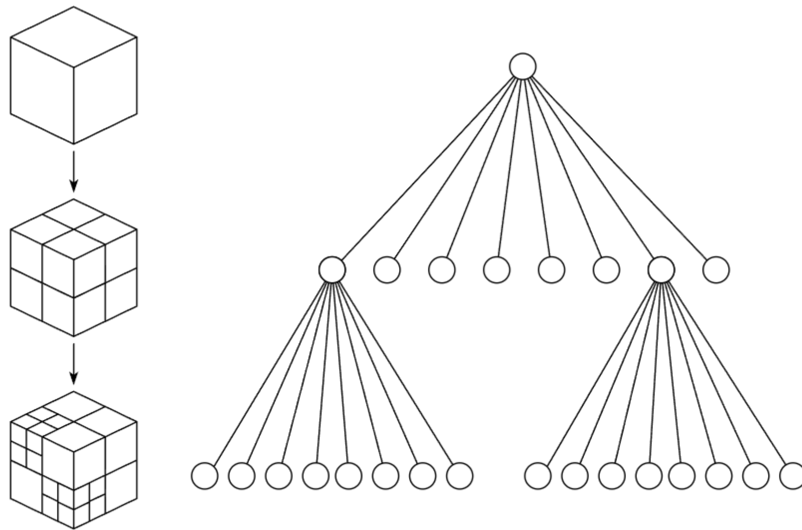
Kuva 3.1: Esimerkki navmesh mallintamisesta. Lähde: [17]. Lisensoitu CC BY-NC-ND 4.0 lisenssillä.

Kuvassa 3.1 on esimerkki navmesh-mallintamisesta. Punainen Agent niminen sylinteri edustaa pelissä olevaa hahmoa. Sininen alue on navmesh alue, jonka sisällä hahmo voi liikkua. Kuva on suositun Unity pelimoottorin dokumentaatiosta. Unity:ssä navmesh on suosittu reitinhakua varten toteutettu mallintamistekniikka, ja pelimoottorissa on kehittäjille valmiita metodeita ja tarvikkeita toteuttaa reitinhakua navmesh mallintamista käyttäen. Valmiita navmesh tarvikkeita on myös muissa suosituissa pelimoottoreissa, kuten Unreal Engine [18].

Navmesh on modernin videopelituotannon standardi tapa toteuttaa 2D-reitinhakua [7]. Vaikka sitä voi käyttää 3D-sovelluksissa, perinteinen navmesh ei kuitenkaan mahdollista täyttä kolmen ulottuvuuden liikettä. Se pelkästään mallintaa kahden ulottuvuuden pintoja, joilla hahmot voivat liikkua. Kirjan [15] luvussa 21 käsitellään lentävien hahmojen navigointia 3D-ympäristöissä. Luvun ensimmäisellä sivulla mainitaan, että sekä säännöllisiä ruudukoita että navmesh voidaan käyttää 3D-ympäristöjen mallintamiseen. Näissä molemmissa on kuitenkin merkittäviä haittapuolia. Säännöllisen ruudukon käyttö 3D-ympäristössä tekee hakutilasta erit-

täin suuren, joka tekee reitinhausta epätehokasta ja vie paljon muistia. Navmesh taas mallintaa ympäristöä pelkästään vaakasuunnassa. Jos halutaan, että hahmo voi myös muuttaa korkeuttaan, täytyy käyttää useita navmesh-pintoja eri korkeuksille. Tämän avulla hahmo voisi vaihdella näiden korkeuksien välillä. [15]

Tilan koon suurentuessa tulee kuitenkin mahdottomaksi määrittää, kuinka monta navmesh-pintaa tulisi käyttää [15]. 3D-mallintaminen on haastava ongelma, johon varsinkin suuremmissa ympäristöissä on usein tarpeellista käyttää tehokkaampaa mallintamista kuin aiemmin mainitut. Yksi vaihtoehto on octree-mallintaminen. Octree on puu-tyyppinen tietorakenne, jossa jokaisella solmulla joko ei ole lapsia tai on kahdeksan lasta. Kuvassa 3.2 on havainnollistava esimerkki octree-mallintamisesta.



Kuva 3.2: Octree-mallintamisen prosessi.

Octree-mallintamisessa suuri hyöty säännölliseen ruudukkoon verrattuna 3D-ympäristöissä on, että solmujen määrä on huomattavasti pienempi. Octree-mallintaminen alkaa siitä, että tila koostuu yhdestä solmusta, jota kutsutaan juurisolmuksi. Jos tilassa on esteitä, niin tila jaetaan kahdeksaan osaan. Näistä osista tulee juurisolmun lapsia. Jos näissä lapsisolmuissa on esteitä, niin ne jaetaan

taas kahdeksaan osaan. Tätä prosessia jatketaan, kunnes jokin solmujen minimikoko tulee vastaan. Tilaa siis tarkennetaan niille osioille, joissa on esteitä. Suuret esteettömät alueet voivat siis koostua yksittäisistä solmuista. Tämän avulla voidaan säästää huomattavasti muistia, sekä tehostaa reitinhakua. Muokattua versiota octree-mallintamisesta käytettiin videopelissä Warframe 3D-reitinhakua varten [15]. [7]

3.2 Heuristiikka

Reitinhakualgoritmeissa heuristiikka on työkalu, jonka avulla algoritmille voi antaa arvion määränpään suunnasta. Tämän avulla algoritmi voi vähentää turhien solmujen tutkimista, ja siten parantaa tehokkuutta. Heuristiikalla tarkoitetaan myös usein heuristisen algoritmin ahnetta luonnetta. Ahneus tarkoittaa sitä, että algoritmi priorisoi laskennallisesti ja ajallisesti halpaa ratkaisua optimaalisen ratkaisun sijaan. Esimerkkinä voi vertailla A* ja Dijkstra algoritmeja. A*-algoritmi käyttää heuristiikkaa. Jos heuristiikan toteuttaa niin, että algoritmi menee käytännössä suoraan määränpäätä kohti, saa tuloksen todennäköisesti hyvin nopeasti. Tämä säästää laskentatehoa ja aikaa. Mutta huono puoli tässä on se, että reitti ei tällöin yleensä ole lyhin mahdollinen. Dijkstran algoritmi taas ei käytä heuristiikkaa. Se etsii määränpäätä tasaisesti joka suunnasta. Löydetty reitti on siis aina lyhin mahdollinen, mutta laskentatehollisesti ja ajallisesti suoritus on kallis. [19]

Jos A*-algoritmia käyttää ilman heuristiikkaa, siitä tulee Dijkstran algoritmi. Mutta jos heuristiikan toteuttaa yliarvioivaksi, siitä tulee ahne. Heuristiikka määrittää siis mitä algoritmi priorisoi suorituksessaan.

Yleisesti videopeleissä pyritään käyttämään algoritmeja, jotka ovat laskennallisesti ja ajallisesti halpoja, ja löytävät tarpeeksi hyvän reitin kyseistä toteutusta varten. Etenkin suurempien hakualueiden toteutuksissa käytetään harvemmin Dijkstra-

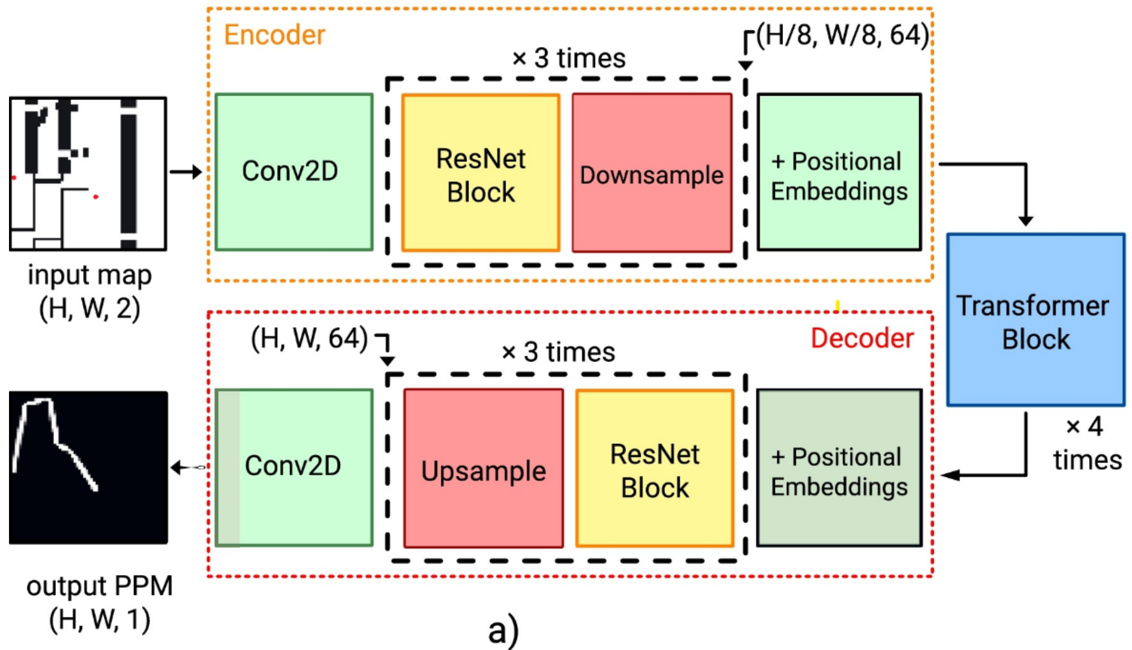
tyyppisiä algoritmeja, sillä ne muuttuvat laskennallisesti ja ajallisesti erittäin kalliiksi hakualueen koon kasvaessa [19].

Artikkelissa [20] käsiteltiin kahden ulottuvuuden platformer-tyyppistä peliä, jossa pelaajan kuuluu päästä tason vasemmasta laidasta oikeaan laitaan. Reitinhakualgoritmia käytettiin luodakseen pelaajalle ns. reittiopas, jonka avulla voi saada selville miten tason läpi pääsee onnistuneesti. Pelaajan liikkeet ovat pystysuunnassa rajoitettuja, sillä peliympäristössä simuloidaan painovoimaa. Lisäksi tasossa voi olla erilaisia ansoja ja esteitä, joita pelaajan täytyy varoa. Reitinhakuun kuuluu siis yksinkertaisen vaakatason liikkumisen lisäksi muita pelaajan tekemiä toimenpiteitä, esim. esteiden yli hyppimistä. Tämä pelitilan monimutkaisuus tekee heuristiikan määrittelyn A*-tyyppisessä algoritmissa vaikeaksi. Artikkelissa heuristiikaksi valittiin pelinsisäinen aika, joka pelaajalla kestää päästä aloituspisteestä määränpäähän. [20]

Äskeisessä kappaleessa käsitelty artikkeli on esimerkki siitä, miten heuristiikkaa voidaan muuttaa peliympäristön ja käyttötarkoituksen pohjalta. Oikean heuristiikan valitseminen on kriittistä reitinhaun tehokkuuden kannalta. Perinteisissä heuristiikkaa käyttävissä reitinhakualgoritmeissa on kuitenkin heikkouksia. Kun heuristiikan avulla arvioidaan, kuinka kaukana solmu on määränpäästä, ei esteitä aina oteta huomioon. Tämä voi johtaa huonoon suorituskykyyn ympäristöissä, joissa on paljon esteitä. Tätä ongelmaa voidaan helpottaa käyttämällä moderneja syväoppivia neuroverkkoja heuristiikkaa varten. Seuraavaksi tarkastellaan miten tämä prosessi toimii. Osio perustuu artikkeliin, jossa käsitellään reitinhakua yleisellä tasolla, eikä suoraan videopeleihin keskittyen. Pohditaan siis myös seuraavassa osiossa, että olisiko tällainen toteutus järkevää oikeassa videopelituotannossa.

3.3 Syväoppimisen käyttö heuristiikan määrittämisessä

Modernien syväoppivien neuroverkkojen avulla voidaan selvittää kyseiselle ympäristölle oikea heuristiikka. Tämä tekee heuristiikasta tarkemman ottamalla huomioon esteet ympäristössä. Tämän avulla voidaan vähentää turhien polkujen tarkastelua, ja siten tehostaa reitinhakua. Syväoppivat neuroverkot voivat selvittää heuristiikan kyseiselle ympäristölle käyttämällä sekä konvoluutioverkkoja että muuntajia. Tämä tehdään ennen itse reitinhakua, niin sanotussa esikäsittelyvaiheessa (engl. preprocessing step). Artikkelissa [13] heuristiikkana käytettiin muun muassa reitin todennäköisyyskarttaa (engl. path probability map). Todennäköisyyskartta kertoo jokaisesta solmusta sen, kuinka todennäköistä on, että solmu on reitillä. Tätä voidaan käyttää, kun liikkumisen hinta solmujen välillä on tiedossa. Lisäksi käytettiin heuristiikkana korjauskerrointa (engl. correction factor). Korjauskertoimessa vertaillaan kyseisestä ympäristöstä riippumattoman heuristiikan, eli perinteisen heuristiikan, ja täydellisen heuristiikan välistä suhdetta. [13]



Kuva 3.3: Todennäköisyyskartan muodostumisen prosessi. Lähde: Mukailtu [13].

Kuvan 3.3 syötteessä (input map) näkyy ympäristö, jossa on paljon esteitä. Syötteessä olevat punaiset pisteet edustavat lähtökohtaa ja määränpäätä. Konvoluutioverkkojen ja muuntajien avulla saadaan syötteestä muodostettua todennäköisyyskartta, joka on prosessin ulostulo (output PPM). Ensimmäinen konvoluutioverkko etsii ympäristöstä lokaaleja ominaisuuksia, kuten esteiden nurkkia. Muuntaja vertailee sitten näitä lokaaleja ominaisuuksia toisiinsa, ja luo globaaleja suhteita niiden välillä. Toinen konvoluutioverkko purkaa muuntajan tuloksen, ja tuottaa lopullisen ulostulon. Tuotettua todennäköisyyskarttaa voidaan sittemmin käyttää toisena heuristiikkafunktiona itse reitinhaussa. Tämä tehostaa reitinhaun toteutusta, sillä hakualue rajautuu hyvin suppeaksi. Itse reitinhakua varten on artikkelissa käytetty Focal Search-algoritmia, kuten luvussa 2.3 mainittiin. Käytännössä hyödynnetään siis kahta heuristiikkaa: todennäköisyyskarttaa ja Focal Search-algoritmin heuristiikkaa. [13]

Todennäköisyyskartan muodostaminen on kuitenkin laskennallisesti vaativa prosessi. Syväoppimismallit täytyy kouluttaa, jotta niitä voidaan käyttää, ja jotta ne tuottavat hyviä tuloksia. Artikkelissa [13] kouluttamisessa kesti 3.5 tuntia jokaista heuristiikkaa kohti, ja se toteutettiin NVIDIA A100 80GB näytönohjaimella, joka on suunniteltu tekoälyn kouluttamista varten. Lisäksi ongelmaa käsiteltiin pelkästään staattisen ympäristön näkökulmasta. Koska syväoppimismenetelmät koulutetaan etukäteen, eivät ne ota huomioon uusia, yllättäen ilmeneviä esteitä. Tämän ongelman voisi kuitenkin ratkaista esimerkiksi käyttämällä Focal Search-algoritmin sijasta luvussa 2.3 esitettyjä algoritmeja, joita käytetään dynaamisissa ympäristöissä.

Pelin kehittäjien kannalta syväoppimismenetelmien kouluttaminen vaatii lisää aikaa ja laskentatehoa. Mutta jos tätä prosessia olisi mahdollista soveltaa luotettavasti oikeisiin videopeliympäristöihin, ja käyttää oikeassa videopelituotannossa, voisi pelien suorituskyky parantua huomattavasti. Pelaajien kannalta syväoppimismenetelmien käytöllä voisi siis olla suuri positiivinen vaikutus pelien vaaditun laskentatehon vähentyessä.

4 Yhteenveto

Johdannossa määriteltiin kaksi tutkimuskysymystä. Ensimmäisessä kysymyksessä pohdittiin, millaisia reitinhakualgoritmeja esiintyy viimeisen viiden vuoden aikana julkaistussa videopelituotantoa käsittelevässä kirjallisuudessa. Ensimmäistä kysymystä käsiteltiin luvussa 2, jossa käytiin läpi erilaisia algoritmeja ja miten ne eroavat toisistaan. Huomattiin, että A*-algoritmi on yhä suosittu kirjallisuudessa, ja että sitä mainitaan usein. 2000-luvun aikana kehitetyt reitinhakualgoritmit perustuvatkin usein siihen. Nämä uudemmat algoritmit ovat usein tehokkaampia kuin perinteinen A*, ja niitä käyttämällä voidaan säästää laskentatehoa ja aikaa. Näistä algoritmeista käsiteltiin LRA*, HPA*, PRA*, JPS sekä Focal Search. Lisäksi käsiteltiin yksittäistä vahvistusoppimiseen perustuvaa algoritmia EPOM.

Toisessa kysymyksessä mietittiin sitä, miten näitä algoritmeja viimeisen viiden vuoden tutkimuskirjallisuuden mukaan sovelletaan videopeleihin. Toista kysymystä käsiteltiin luvussa 3, jossa esiteltiin erilaisia mallintamistekniikoita, ja miten niitä voi käyttää reitinhakua varten erilaisissa peliympäristöissä. Käsiteltiin myös heuristiikkaa, ja miten se vaikuttaa reitinhaun prosessiin. Lopuksi tutkittiin myös syväoppimisen mahdollista käyttöä heuristiikan määrittämisessä. Säännölliset ruudukot, navmesh ja octree ovat yleisesti käytettyjä mallintamistekniikoita, joiden avulla sovelletaan reitinhakualgoritmeja erilaisiin ympäristöihin. Lisäksi käytetään erilaisia heuristiikkoja tehostamaan reitinhakua.

Tässä tutkielmassa reitinhakualgoritmeja on käsitelty saatavilla olevan kirjallisuuden avulla. Todellisen videopelituotannon toteutuksista on vaikeaa saada tietoa. Pelejä kehittävät usein suuret kaupalliset yritykset, jotka eivät yleensä paljasta julkisuudelle millaisia tekniikoita heillä on käytössä. Oikeista toteutuksista on kuitenkin saatavilla jotain tietoa, esimerkiksi artikkelista [21] tai kirjasta [15]. Dragon Age: Origins [10] ja Warframe [15] ovat esimerkkejä suosituista peleistä, joissa on käytetty tässä tutkielmassa käsiteltyjä tekniikoita.

Reitinhakualgoritmien jatkuva kehittyminen tulee olemaan kriittistä monen teknillisen alan kannalta. Erilaiset autonomiset järjestelmät, karttasovellukset ja videopelit ovat osa-alueita, joissa kysyntä ja markkinat kasvavat jatkuvasti. Ajankohtaisena esimerkkinä voi käyttää Nasan Mars-mönkijää Perseverance, joka on tutkinut Marsia vuodesta 2021 lähtien. Perseverance käyttää muokattua versiota Dijkstran algoritmista suunnitellakseen reittejä [22]. Tämä on esimerkki siitä, miten reitinhakutekniikoiden tutkiminen ja parantaminen voi tuoda paljon hyötyä monenlaisiin eri toteutuksiin. Tulevat tutkimukset voisivat perehtyä esimerkiksi syvä- ja vahvistusoppimisen parannettuun käyttöön videopelien reitinhaussa, heuristiikan parantamiseen eri käyttötarkoituksia varten tai uusien mallintamistekniikoiden kehittämiseen.

Lähdeluettelo

[1] A. Elshahed, M. K. Bin Majahar Ali, A. S. A. Mohamed, F. A. B. Abdullah ja T. L. J. Aun, "Efficient Pathfinding on Grid Maps: Comparative Analysis of Classical Algorithms and Incremental Line Search", *IEEE Access*, vol. 13, s. 98 473–98 484, 2025. DOI: 10.1109/ACCESS.2025.3575168.

[2] D. Foead, A. Ghifari, M. B. Kusuma, N. Hanafiah ja E. Gunawan, "A Systematic Literature Review of A* Pathfinding", *Procedia Computer Science*, vol. 179, s. 507–514, 2021, ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2021.01.034.

[3] A. Afzalov, A. Lotfi ja M. E. Aydin, "A Strategic Search Algorithm in Multi-agent and Multiple Target Environment", teoksessa *RiTA 2020*, E. Chew et al., toim., Singapore: Springer Singapore, 2021, s. 195–204, ISBN: 978-981-16-4803-8. DOI: https://doi.org/10.1007/978-981-16-4803-8_21.

[4] P. E. Hart, N. J. Nilsson ja B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths", *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, nro 2, s. 100–107, 1968. DOI: 10.1109/TSSC.1968.300136.

[5] D. Silver, "Cooperative pathfinding", *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 1, nro 1, s. 117–122, 2005, ISSN: 2334-0924, 2326-909X. DOI: 10.1609/aiide.v1i1.18726.

[6] A. Botea, M. Muller ja J. Schaeffer, ”Near optimal hierarchical path-finding”, *J. Game Dev.*, 2004.

[7] Q. Massonnat ja C. Verbrugge, ”Efficient Octree-based 3D Pathfinding”, teoksessa *2024 IEEE Conference on Games (CoG)*, 2024, s. 1–8. DOI: 10.1109/CoG60054.2024.10645669.

[8] A. Afzalov, A. Lotfi, B. Inden ja M. E. Aydin, ”A Strategy-Based Algorithm for Moving Targets in an Environment with Multiple Agents”, *SN Computer Science*, vol. 3, nro 6, s. 435, 2022, ISSN: 2661-8907. DOI: 10.1007/s42979-022-01302-x.

[9] N. Sturtevant ja M. Buro, ”Partial pathfinding using map abstraction and refinement”, teoksessa *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 3*, sarja AAAI’05, Pittsburgh, Pennsylvania: AAAI Press, 2005, s. 1392–1397, ISBN: 1-57735-236-X. url: https://dl.acm.org/doi/abs/10.5555/1619499.1619557.

[10] N. Sturtevant, ”Memory-efficient abstractions for pathfinding”, *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 3, nro 1, s. 31–36, 2007, ISSN: 2334-0924, 2326-909X. DOI: 10.1609/aiide.v3i1.18778.

[11] D. Harabor ja A. Grastien, ”Online graph pruning for pathfinding on grid maps”, *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 25, nro 1, s. 1114–1119, 2011, ISSN: 2374-3468, 2159-5399. DOI: 10.1609/aaai.v25i1.7994.

[12] S. K. Moghadam, M. Ebrahimi ja D. D. Harabor, ”Guards: Benchmarks for weighted grid-based pathfinding”, *Expert Systems with Applications*, vol. 249, s. 123 719, 2024, ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2024.123719.

[13] D. Kirilenko, A. Andreychuk, A. I. Panov ja K. Yakovlev, ”Generative models for grid-based and image-based pathfinding”, *Artificial Intelligence*, vol. 338, s. 104 238, 2025, Figure Reprinted with permission from Elsevier, ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2024.104238.

[14] A. Skrynnik, A. Andreychuk, K. Yakovlev ja A. I. Panov, ”When to Switch: Planning and Learning for Partially Observable Multi-Agent Pathfinding”, *IEEE Transactions on Neural Networks and Learning Systems*, vol. 35, nro 12, s. 17 411–17 424, 2024. DOI: 10.1109/TNNLS.2023.3303502.

[15] Rabin, Steve, *Game AI Pro 3 : Collected Wisdom of Game AI Professionals*. CRC Press LLC, 2017, ISBN: 9781498742597.

[16] Z. Abd Algfoor, M. S. Sunar ja H. Kolivand, ”A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games”, *International Journal of Computer Games Technology*, vol. 2015, nro 1, 2015. DOI: https://doi.org/10.1155/2015/736138.

[17] Unity Technologies. ”Unity Documentation”. Lisenssi: https://creativecommons.org/licenses/by-nc-nd/4.0/deed.en, viitattu 10. joulukuuta 2025. url: https://docs.unity3d.com/Packages/com.unity.ai.navigation@2.0/manual/NavInnerWorkings.html.

[18] Epic Games. ”Unreal Engine Documentation”, viitattu 10. joulukuuta 2025. url: https://dev.epicgames.com/documentation/en-us/unreal-engine/basic-navigation-in-unreal-engine.

[19] K. C. Ugwoke, N. A. Nnanna ja S. E.-Y. Abdullahi, ”Simulation-based review of classical, heuristic, and metaheuristic path planning algorithms”, *Scientific Reports*, vol. 15, nro 1, s. 12 643, 2025, ISSN: 2045-2322. DOI: 10.1038/s41598-025-96614-2.

[20] R. Bishop ja D. Churchill, ”The Effects of Human-like Modifications to Heuristic Action Evaluation in Video Game Pathfinding”, teoksessa *Proceedings of the 17th International Conference on the Foundations of Digital Games*, sarja FDG '22, New York, NY, USA: Association for Computing Machinery, 2022, ISBN: 978-1-4503-9795-7. DOI: 10.1145/3555858.3555888.

[21] J. Pfau, J. D. Smeddinck ja R. Malaka, ”The Case for Usable AI: What Industry Professionals Make of Academic AI in Video Games”, teoksessa *Extended Abstracts of the 2020 Annual Symposium on Computer-Human Interaction in Play*, sarja CHI PLAY '20, event-place: Virtual Event, Canada, New York, NY, USA: Association for Computing Machinery, 2020, s. 330–334, ISBN: 978-1-4503-7587-0. DOI: 10.1145/3383668.3419905.

[22] V. Rajesh ja C. Q. Wu, ”An Extension of Pathfinding Algorithms for Randomly Determined Speeds”, teoksessa *2024 IEEE International Performance, Computing, and Communications Conference (IPCCC)*, 2024, s. 1–8. DOI: 10.1109/IPCCC59868.2024.10850316.